An Architecture for Improving Variable Radix Real and Complex Division Using Recurrence Division

James E Stine, Miloš D Ercegovac, Jean-Michel Muller

An Architecture for Improving Variable Radix Real and Complex Division Using Recurrence Division

James E. Stine
School of Electrical and
Computer Engineering
Oklahoma State University
Stillwater, Oklahoma 74078

Miloš D. Ercegovac
Computer Science Department
University of California at Los Angeles
Los Angeles, California, USA

Jean-Michel Muller
CNRS-Laboratoire LIP
(CNRS, ENS Lyon, Inria, UCBL)
Ecole Normale Supérieure de Lyon
46 Allee d'Italie,
69364 Lyon Cedex 07, France

***Abstract*—This paper shows the details of an implementation of variable radix floating-point complex division based on previous implementations of the algorithm. This implementation takes advantage of the easier prescaling offered by low-radix division and recodes it as necessary for higher radix iterations throughout the design. This, along with proper use of redundant digit sets, allows us to significantly altar performance characteristics relative to exclusively high-radix division implementations. Comparisons to existing architectures are shown, as well as common implementation optimizations for future iterations. Results are given in cmos32soi 32nm MTCMOS technology using ARM-based standard-cells and commercial EDA toolsets.**

I. Introduction

As silicon manufacturing technology gets ever smaller feature sizes, the corresponding increased number of transistors requires ever more power to drive devices [1], [2]. Furthermore, special-purpose and application-specific hardware has placed further demands on processor in terms of operation, speed and energy costs [3]. Moreover, specialized arithmetic is also needed to handle higher performance at a lower power for applications such as signal processing and Internet of Things (IoT) [4]. This is evident for operations such as division that can dramatically impact energy and performance [5]. Looking forward, new architectures will have to redesign existing division and other arithmetic operations with more power-efficient techniques as more computation grows within hardware [6].

Division can be computed in a varied number of ways, however, many current architectures tend to perform multiplicative-based division and square root by sharing it with a multiplier [5]. One method that is particularly well suited to handling division is by using a recurrence for the division operation [7]. More importantly, recurrence methods are typically lower in energy per digit than multiplier-based architectures [8].

Real and complex division architectures have, historically, fallen into few categories. Algorithms such as SRT division and similar designs based on a recurrence, typically give smaller die footprints and lower power consumption [8]. This fixes one of the primary issues for high radix division, but the hardware requirements of SRT division, and algorithms like it, are prohibitively large. Faster algorithms and architectures exist, such as those based on Newton-Raphson iterations or series expansions [9], [10], but since these designs use large multipliers, power consumption and area can easily become unmanageable for real-world constraints. Some of these drawbacks can be mitigated either directly with expanded look-up tables, prescaling tables and postscaling that can be used for higher radix architectures [11]. Implementing these for very large radices, however, can be nearly impossible in practice, due to the considerable size required for these tables.

Variable radix designs fix this problem by dynamically changing the effective prescaling as the operation occurs. Initially, a small prescaling value is applied at the beginning of an operation. And, a prescaling table is utilized, the design is small enough to be easily implemented and managed. Once sufficient precision is achieved, the partial remainder is scaled again to a higher radix. This prescaling can occur repeatedly to create architectures that produce an otherwise unachievable ultra-high radix output, such as radix 256, in a comparable number of cycles to more standard division architectures at a much lower radix.

This paper demonstrates the implementation of variable radix architecture based on [12]. This architecture uses a variable radix prescaling methodology and replaces a significant amount of its datapath with a redundant scheme for both partial remainder and quotient generation. Left-to-Right (LR) multiplication [13] is used for faster carry generation for redundant signals, and rounding using on-the-fly conversion [14] is also implemented on the final variable radix stage for this architecture to reduce delay.

Synthesis results use a cmos32soi 32nm MTCMOS technology with ARM-based standard cells. The combination of design improvements over standard architectures yields competitive results for ultra-high radix implementations, especially in terms of power consumption and area used. Although this method can be applied here to both real and complex division, the implementation results are only applied to 32-bit complex division as a proof of concept. Future work can be easily done combining both units into one system for minimal impact and lower energy operations per digit as well as higher input operand widths.

II. Background

Digit-recurrence division [7] uses a recursive equation where $w[j]$ is the partial remainder for iteration j, d is the divisor, r is the radix, and $q[j]$ is the quotient digit for iteration j:

$$w[j+1] = r \cdot w[j] - d \cdot q[j+1] \ .$$

The quotient selection function is based on comparisons between low-precision estimates of the divisor and the shifted partial remainder.

$$q[j+1] = QST(r \cdot \widehat{w[j]}, \widehat{d}) \ ,$$

where QST is the Quotient Selection Table [7]. The QST can be implemented using different methodologies including ROM tables, PLAs, and combinational logic [7].

To reduce the complexity of the quotient-digit selection and the generation of the divisor multiples, a signed-digit notation is typically utilized for the quotient. Different levels of redundancy can be also utilized where the maximum digit a is such that $a \leq r-1$ to avoid having numbers that are overredundant [7]. The selection of a ultimately influences the complexity of the hardware.

A prominent advantage of utilizing recurrence methods for algorithms such as division is that they can also be combined with other key algorithms at low cost, such as convolution, video processing and machine learning giving rise to new computing architectures [6]. More importantly, recurrence methods for division and square root have significant advantages over other methods, such as other multiplicative-based methods in that it is much easier to get a correctly-rounded result [15]. And, by using a redundant number systems [16], the QST becomes simpler and easier to implement than other division-based methods [7]. With a larger radix, fewer iterations are needed; unfortunately the QST grows exponentially with the radix.

An elegant method for reducing the complexity of the QST is to employ prescaling of the operands [17]. Prescaling techniques makes the quotient-digit selection function *independent* of divisor d by thus reducing its complexity. That is, prescaling works by multiplying the numerator and divisor via shifting and adding so that its value is closer to its maximum value (e.g., $d \approx 1$) to simplify the QST. Although this works relatively well for low radices, as the radix increases more accurate prescaling factors are needed to allow the correct computation. This ultimately makes prescaling for a given radix difficult as it imposes unnecessary memory requirements to store the correct prescaling factors.

A. Complex Division by Recurrence

Complex division is an important application in the scientific and engineering community [15]. A straightforward method to implement complex division is to use several multiplications and additions [15]

$$\frac{n_R + i \cdot n_I}{d_R + i \cdot d_I} = \frac{n_R \cdot d_R + n_I \cdot d_I + i \cdot (n_I \cdot d_R - n_R \cdot d_I)}{d_R^2 + d_I^2} \ ,$$

where $n = n_R + i \cdot n_I$ is the numerator or dividend, $d = d_R + i \cdot d_I$ is the denominator or divisor. For this research, the numerator and denominator are assumed to be between $0 \leq n, d < 1$ although it can be computed between $[1, 2)$ by shifting. There is additional discussion in literature in utilizing different ordering of operations using fixed-size arithmetic for complex division to avoid multiple combinations of arithmetic that can potentially overflow [18], [19]. Consequently, methods that implement brute-force methods based on the equations above can result in non-optimal or not practical implementations [20].

As for, real division, an efficient and novel method to handle complex division in hardware is by using digit recurrence and prescaling in an intelligent way [15]. This works by using the recurrence relation for division shown earlier where $w[j]$ is the j-th partial remainder, $w[0] = n = (nR, nI)$ is the dividend, $d = (dR, dI)$ is the divisor, and $q_{j+1} = (qR_{j+1}, dI_{j+1})$ is the quotient digit obtained in the j-th iteration.

Prescaling computations of the dividend and divisor are computed before the operation starts with the scaling factor $K = K1 + i \cdot K2$. This produces the following complex prescaling factors such that

$$\begin{array}{rcrcl} yR & = & dR \cdot K1 & - & dI \cdot K2 \ , \\ yI & = & dI \cdot K1 & + & dR \cdot K2 \ , \\ xR & = & nR \cdot K1 & - & nI \cdot K2 \ , \\ xI & = & nI \cdot K1 & + & nR \cdot K2 \ . \end{array}$$

Each of the recurrences contains one additional term compared to the conventional digit-recurrence division algorithm producing the following two recurrences where (xR, xI) and (yR, yI) are the prescaling factors:

$$\begin{array}{rcl} wR[j+1] & = & r \cdot wR[j] - qR[j+1] \cdot yR + qI[j+1] \cdot yI \\ wI[j+1] & = & r \cdot wI[j] - qI[j+1] \cdot yR - qR[j+1] \cdot yI \end{array} \ .$$

As mentioned earlier, a key advantage to these digit-recurrence methods is that obtaining a correctly rounded result is straightforward and much easier hardware wise than more complex methods that use a multiplier. However, careful attention has to be placed to the prescaling values as well as the number of bits required to select the quotient to compute the result in the correct number of cycles [15]. For recurrence methods, the number of cycles is:

$$n_{cycles} = \left\lceil \frac{Q[length]}{log_2 \ r} \right\rceil \ ,$$

where $Q[length]$ denotes the size of the quotient for a given recurrence.

In [15], the algorithm is designed so that is used to select the correct quotient digits given the radix and prescaling factor. The requirements to satisfy a correct result is given as:

$$2^{-p+1} \cdot a + \frac{1}{2} + 2^{-\sigma} \leq \Omega = \frac{1}{r} \cdot (a + \frac{1}{2} - 2^{-\sigma}) \ ,$$

where p, σ, and Ω are chosen based on the values of r and a. In order to simplify the prescaling operation, the prescaling factor $(1/\hat{d})$ is calculated and rounded to a reduced precision.

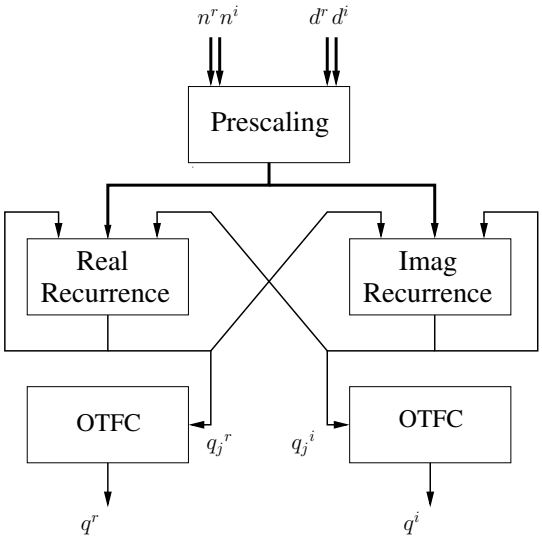

Fig. 1. Block Diagram of Complex Division using Recurrence.

For example, for $r = 8$, $a = 15$, one can choose $\sigma = 4$ and $p = 8$. As mentioned in [15], there is an inverse relationship between p and a potentially affecting the hardware for prescaling and subsequently its implementation. For example, for 32-bit input operands of $n = (0.359308 - i \cdot 0.922485)$ and $d = (0.996379 + i \cdot 0.986769)$, the prescaling factors are `K1 = 0x82` and `K2 = 0x80` produce a result of $q = (-0.280846 - i \cdot 0.6576796)$ with 36.819 bits of accuracy in 9 cycles satisfying the precision of the input operands.

III. Implementation

Implementations of recurrence equations are simplified by using redundant notations within the hardware, such as carry-save logic [7]. The basic block diagram of the logic looks like Figure 1 Although combinational approaches have been utilized for recurrence architectures, this work uses sequential-logic with registers. Since the result digits are in redundant form in Figure 1, the quotient must be converted into conventional form. Therefore, on-the-fly conversion units are utilized after n_{cycles} to convert and round the quotient appropriately. On-the-fly conversion (OTFC) is an innovative way to convert the quotient from redundant to conventional form without a carry-propagate adder and involves only shifting and concatenation [21].

The implementation of this complex architecture is efficient but requires large amounts of resources for prescaling especially for higher radix implementations. Table I shows sizes that are required for prescaling the dividend and divisor using exhaustive techniques. Techniques in [22] and [23] have utilized some ancillary logic and interpolation, respectively, to reduce the memory requirements. Unfortunately, interpolation methods may take longer to compute the prescaling step as it takes longer to interpolate the correct value. Bipartite methods can also be utilized at the expense of some additional logic, as suggested in [15]. Despite these advancements in optimization, a large amount of memory is required just to handle prescaling for larger radices. The value of p is utilized to determine the memory requirements for prescaling.

size	Memory Size	Total (bits)
4	4×8	256
8	12×8	128 KiB
16	28×16	16 GiB

TABLE I
PreScaling Lookup Memory Sizes

A. Variable Radix Division by Recurrence

One novel way to prevent issues related to the size in precaling is to use a method discussed in [12]. Variable radix methods use lower radix methods to start for recurrence, since the scaling factor M is an approximation to $1/d$. After a few iterations of a low-radix division, called "preliminary" in the original work [12], M is then utilized in the higher radix after a specific number of iterations. This simplifies the prescaling of the operands since Table I shows that with low radices, prescaling tables are easily feasible.

This paper utilizes the values given in [12] where radix 4 iterations are utilized initially followed by radix 16 iterations, and finally radix 256 iterations. It is important to understand that the architecture for recurrence is built from higher radix operations (i.e., in this case $r = 256$). Since $r = 2^2$, 2^4, 2^8 are all powers of two, the hardware can be shared during the recurrence without harming the intermediate results.

The block diagram of the design is shown in Figure 2. The RR module produces the five radix 4 digits by rounding and recoding the recurrence from the main unit. This is basically a multiplexer that chooses between different number of digits during each radix change. Instead of utilizing four different adders and multiplexors to handle the scaling a left-to-right (LR) [13] multiple generator is utilizing to make things easier for conversion between radices

The LR multipliers utilized in this architecture are designed to handle the scaling in an efficient manner and generate results in a redundant notation that can be utilized for the main recurrence. Therefore, the LR multipliers do not use carry-propagate adders (CPA) and deploy a carry-save structure within the implementation. Additionally, partial-product reduction mechanisms are utilized within the block to reduce the scaled values down to obtain the correct implementation. Each LR multiplier also generates the correct redundant notation and multiple generations accordingly. Again, the OTFC unit is utilized to convert the quotient into its conventional form.

The method shown in Figure 2 is shown only for real division. The basic recurrence block is repeated to exchange values between each other for the real and imaginary blocks as in Figure 1. As stated previously, this unit could conceivably be utilized for a combined division and complex division unit with some additional hardware.

We assume 53-bit input operands so that our operator could potentially be utilized for a floating-point unit. A maximally redundant notation ($a = r - 1$) is utilized within the blocks to reduce the complexity of hardware. The implementation

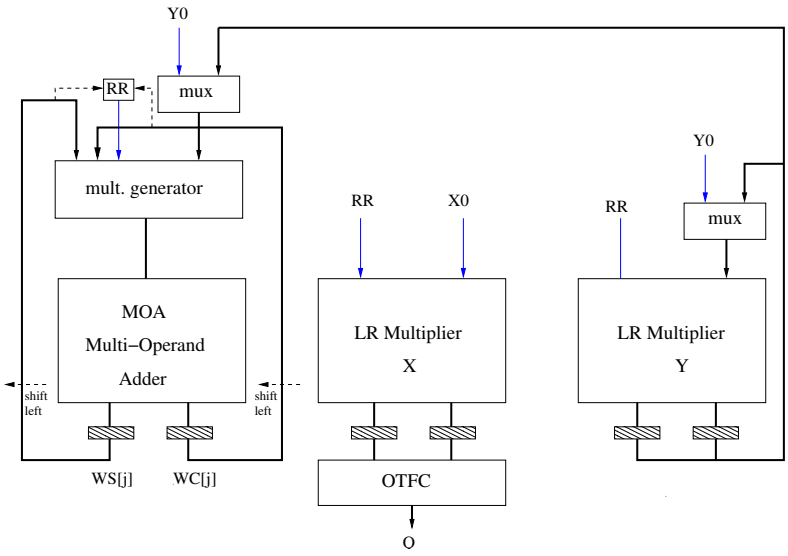

Fig. 2. Block Diagram of Conventional Division using Variable Radix Approach.

presented here also requires several iterations to achieve a result that is accurate enough for each prescaling. This results in the following sequence in operation: $R4 \rightarrow R16 \rightarrow R256$ where the RR block in Figure 2 shifts the partial remainder based on the appropriate radix. A finite-state machine (FSM) is incorporated into the design to make sure the unit completes the appropriate number of steps and controls the datapath accordingly. One iteration is utilized at the beginning of the recurrence to determine the prescaling values. Afterwards, 12 iterations, the unit completes approximately 56 bits of precision for a given 53-bit input operand.

IV. Results

The proposed design is implemented in RTL-compliant System Verilog and designs are then synthesized using an ARM 32nm CMOS library for Global Foundries (GF) cmos32soi technology optimizing on delay. Where appropriate, Verilog is written to take advantage of any Intellectual Property (IP) through Synopsys ® DesignWare (DW) (e.g., `assign Sum=A+B`). To verify the correctness, all implementations are tested against several thousand random test vectors generated by a MATLAB program. MATLAB is utilized heavily in this work to provide good verification of the results at each step of the iteration so that it was easier to debug the implementation.

An ARM standard-cell library is utilized with multiple values of V_T to aid in synthesis (i.e., MTCMOS). Synthesis is optimized for delay utilizing Synopsys® (SNPS) Design Compiler™ (DC) in topographical mode using a PVT process at 25° C using TT corners. Topographical synthesis or physical synthesis is provided by Synopsys® DC™ (DC) to accurately predict timing, area and power by including information from the standard-cell layouts and underlying interconnect. The fanout-of-4 (FO4) delay is a good metric to give comparisons of designs compared to other methods. FO4 is unitless and gives a good delay regardless of process, supply voltage and temperature (PVT) [1]. The average fanout-of-4 (FO4) delay measured with SPICE for this GF cmos32soi technology is measured at 5.95ps.

All designs are synthesized at 1 GHz in order to give fair comparisons between each algorithm. This frequency allows all designs to meet the necessary speed constraint and not be inflated due to non-constraining paths. Table II shows the post-synthesis results with the cmos12soi IBM/GF 32nm CMOS technology using the Synopsys® DC™ synthesis software. Results show a significant drop in area and power compared to multiplication-based methods for division with 53 bits (-50.92% area and -53.21% power).

Energy and power results are obtained in Table II by use of vector history. Approximately $1,000$ vectors were input into the architecture via a Value Change Dump (VCD). VCD files are converted to SAIF files and imported into Synopsys DC™ to produce the energy results. Static and Leakage are combined into column within Table II and dynamic power is inferred from the subtraction of Total − Static/Leakage.

The design are compared against several division designs written for this work. This includes a design found in [15], a conventional $r = 64$ division by recurrence unit using overlapping, and Goldschmidt's algorithm for division. The Goldschmidt unit utilizes a simple lookup for its initial seed to avoid excessive area for memory and utilizes a Booth multiplier within the multiplier.

Division by recurrence architectures do not converge as quickly as other multiplicative-based methods for division, however, they consume significantly smaller amounts of area and potentially produce significant savings in energy [7]. Goldschmidt's algorithm quadratically converges to the final quotient and although it takes 5 iterations to compute the result with a 2-bit initial approximation, only 1 multiplier is utilized within this paper. Consequently, it takes $5 \times 2 = 10$ cycles to

TABLE II
PHYSICAL SYNTHESIS IMPLEMENTATION RESULTS IN 32NM

Division Architecture			Power [mW]	
	Cells	Area [um^2]	Static/Leakage	Total
Variable Complex (this work)	6,683	5,301.3839	2.930	4.019
Complex $r = 8$ using [15] approach	5,030	4,301.0331	2.703	3.930
$r = 64$ division with overlapping	4,328	3,806.3610	2.537	3.228
Goldschmidt's algorithm	8,140	8,764.0019	4.218	8.400

converge or iterate to the correct result. Although pipelining can be utilized for the multiplier to reduce the number of cycles, a single-cycle multiplier is employed to compute each iteration.

Comparing this work to [15], this work consumes some additional area mainly due to the LR multiplier units. However, the implementation in Table II for [15] is only for $r = 8$ compared to this work that eventually goes up to $r = 256$. The real division unit in Table II is a $r = 4$ unit that utilizes overlapping to increase its radix to $r = 64$. The real division unit is obviously smaller than the two complex units in that it does not perform complex division. It is also worth noting that the Goldschmidt unit shown in Table II only performs division and does not compute complex division.

V. SUMMARY

This paper demonstrates that with variable radix one can make prescaling realizable for high-radix division. The results indicate a dramatic decrease in area and energy compared to a division unit that employs multiplication. Prescaling is beneficial in lowering QST complexity especially with the use of a variable-radix unit. Simple additions of hardware can be made to this unit to allow it to compute both conventional and complex division. In addition, with additional optimizations in HDL coding, perhaps using new UPF-style constructs, further reduction can occur in energy and power. In summary, one can avoid the demand for rather large prescaling tables by starting with a small radix and gradually increasing it.

REFERENCES

[1] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. USA: Addison-Wesley Publishing Company, 2010.

[2] A. Kumar, L. Shang, L. Peh, and N. K. Jha, "System-level dynamic thermal management for high-performance microprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 96–108, Jan 2008.

[3] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833–854, 1997.

[4] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.

[5] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Transactions on Computers*, vol. 46, no. 2, pp. 154–161, 1997.

[6] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar 2018.

[7] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Norwell, MA, USA: Kluwer Academic Publishers, 1994.

[8] M. D. Ercegovac and J. E. Stine, "Conditional estimation of residuals with prescaling for use in low-energy division units," in *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, 2019, pp. 603–607.

[9] M. Joldes, J. M. Muller, and V. Popescu, "On the computation of the reciprocal of floating point expansions using an adapted newton-raphson iteration," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014, pp. 63–67.

[10] T. Kwon, J. Sondeen, and J. Draper, "Floating-point division and square root implementation using a taylor-series expansion algorithm," in *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, 2008, pp. 702–705.

[11] Taek-Jun Kwon and J. Draper, "Floating-point division and square root implementation using a taylor-series expansion algorithm with reduced look-up tables," in *2008 51st Midwest Symposium on Circuits and Systems*, 2008, pp. 954–957.

[12] M. D. Ercegovac and J. M. Muller, "Variable radix real and complex digit-recurrence division," in *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, 2005, pp. 316–321.

[13] M. D. Ercegovac and T. Lang, "Fast multiplication without carry-propagate addition," *IEEE Transactions on Computers*, vol. 39, no. 11, pp. 1385–1390, 1990.

[14] ——, "On-the-fly rounding for division and square root," in *Proceedings of 9th Symposium on Computer Arithmetic*, 1989, pp. 169–173.

[15] M. D. Ercegovac and J. . Muller, "Complex division with prescaling of operands," in *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. ASAP 2003*, 2003, pp. 304–314.

[16] A. Avizienis, "Signed-digit numbe representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389–400, Sep. 1961.

[17] M. D. Ercegovac and T. Lang, "Simple radix-4 division with operands scaling," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1204–1208, Sep. 1990.

[18] R. L. Smith, "Algorithm 116: Complex division," *Commun. ACM*, vol. 5, no. 8, p. 435, Aug. 1962. [Online]. Available: https://doi.org/10.1145/368637.368661

[19] M. D. Ercegovac and J.-M. Muller, "Design of a complex divider," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XIV*, F. T. Luk, Ed., vol. 5559, International Society for Optics and Photonics. SPIE, 2004, pp. 51 – 59. [Online]. Available: https://doi.org/10.1117/12.560154

[20] J. Liu, B. Weaver, and Y. Zakharov, "Fpga implementation of multiplication-free complex division," *Electronics Letters*, vol. 44, no. 2, pp. 95–96, 2008.

[21] M. D. Ercegovac and T. Lang, "On-the-fly rounding (computing arithmetic)," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1497–1503, 1992.

[22] D. Wang, N. Zheng, and M. D. Ercegovac, "A radix-8 complex divider for fpga implementation," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 236–241.

[23] D. Wang and M. D. Ercegovac, "A radix-16 combined complex division/square root unit with operand prescaling," *IEEE Transactions on Computers*, vol. 61, no. 9, pp. 1243–1255, 2012.